

C13

IsDebugOn	2	(EDMDispatch.c)
daemon_become_daemon.....	10	(EDMDispatch.c)
daemon_catch_interrupts	5	(EDMDispatch.c)
daemon_check_proper_ID.....	7	(EDMDispatch.c)
daemon_cleanup	18	(EDMDispatch.c)
daemon_initialize_logging.....	9	(EDMDispatch.c)
daemon_specific_initialization	16	(EDMDispatch.c)
display_usage.....	4	(EDMDispatch.c)
kill_handler	3	(EDMDispatch.c)
parse_commandline.....	8	(EDMDispatch.c)
rpc_init	12	(EDMDispatch.c)
rpc_run.....	15	(EDMDispatch.c)


```
1  /*
2  ** Copyright 1996,1997 EMC Corporation
3  */
4
5  /*
6  ** EDMDispatch.c
7  **
8  ** Mission Statement: This is the main service file for the dispatch
9  **                    daemon.
10 **                    This file contains the callbacks from the main
11 **                    function
12 **                    which prepares the daemon to go off and service
13 **                    RPC's.
14
15 ** Primary Data Acted On:
16
17 ** Compile-Time Options:
18
19 ** None.
20
21 ** Basic idea here: Module for UNIX specific daemon initialization
22 */
23
24 /* The following provides an RCS id in the binary that can be located
25 ** with the what(1) utility. The intent is to keep this short.
26 */
27 #if !defined(lint)
28 static char RCS_id [] = "@(#)SRCfile: EDMDispatch.c,v $ "
29 "Revision: 1.23 $ "
30 "Date: 1997/02/06 20:49:15 $" ;
31 #endif
32
33 /* #define _POSIX_SOURCE  unable to compile with this define set */
34 /* #define _XOPEN_SOURCE  unable to compile with this define set */
35
36 #include <esl/c_portable.h>
37
38 #include <esl/ep_xopen.h>
39
40 #include <esl/inout.h>
41
42 #include <stdarg.h>
43 #include <string.h>
44 #include <syslog.h>
45 #include <pthread.h>
46 #include <sys/resource.h>
47
48 #include <logging/logging.h>
49 #include <util/esl_core.h>
50 #include <util/esl_pidfile.h>
51 #include <util/esl_daemon.h>
52 #include <csc/csccomm.h>
53
54 #include <restore/csc_EDMDispatch.h>
55
56 #include <EDMmain.h>
57 #include <EDMD_cw.h>
58 #include <EDMD_dispatchlog.h>
59 #include <EDMD_dispatchbackground.h>
60 #include <EDMDcr_rtsvc.h>
61
62 /*
63 * Need to define _XOPEN_SOURCE for signal funtion definitions
64 * and certain signal structure definitions.
65 */
66 #define _XOPEN_SOURCE
67
68 #define EDMDispatch.c 1
```

```
65 #include <signal.h>
66
67 #undef _XOPEN_SOURCE
68
69 static rpc_if_handle_t if_spec;
70
71 static int G_debug = FALSE;
72 /* Variable which will disable forking */
73
74 static char **commandlineargs; /* Pointer to command line args */
75
76 /*****
77 **
78 ** Routine: IsDebugOn
79 **
80 ** Inputs: None
81 **
82 ** Outputs: None
83 **
84 ** Return Codes:
85 ** TRUE if debug is on.
86 **
87 ** Purpose: This routine can be used to tell other subsystems
88 **          whether debugging is available.
89 **
90 ** Intended caller: internal only.
91 *****/
92
93 boolean_t
94 IsDebugOn()
95 {
96     return G_debug;
97 }
```

```
99  /*****
100  **
101  ** Routine: kill_handler
102  ** Inputs: int signal - the signal which was received.
103  **
104  ** Outputs: Will log messages telling what action is being taken.
105  **
106  ** Return Codes:
107  ** exits with the number of the signal received
108  **
109  ** Purpose: This routine handles specific signals i.e. SIGINT,
110  **          SIGQUIT,
111  **          SIGTERM. Each results in a log entry and an exit.
112  **
113  ** Intended caller: internal only.
114  **
115  *****/
116  static void kill_handler( IN int signal )
117  {
118  int status;
119  time_t current_time;
120  char *ctimebuf;
121  char *ebuff;
122  }
123  /* If main exits, it calls this routine with signal 0 */
124  1
125  1
126  1 /* Unregister the interface */
127  1 (void) csc_unregister_server_interface(&if_spec, &status);
128  1
129  1 /* If the unregister fails, report the problem, but continue */
130  1 if ( status != error_status_ok )
131  2 {
132  2 ebuff = (char *) csc_get_error( status );
133  2
134  2 (void) EDMDispatch_logent(
135  2 FILE_, LINE_, LOG_ERR, MESSAGE_NO_LOGIN, 0,
136  2 "CSC_SERVER_LOGIN failed: <td> %s",
137  2 status, (ebuff ? ebuff : "Unknown error") );
138  2 }
139  1 /* Get the current time */
140  1 (void) time(&current_time);
141  1
142  1 ctimebuf = ctime(&current_time);
143  1
144  1 /* Overlay newline with null - buf should always be 26 bytes long */
145  1 ctimebuf[ strlen(ctimebuf) - 1 ] = 0;
146  1
147  1 (void) EDMDispatch_logent(
148  1 FILE_, LINE_, LOG_INFO, MESSAGE_SHUTDOWN, 0,
149  1 "Shutting down at %s due to signal %d", ctimebuf,
150  1 signal);
151  1
152  1 /* Remove our lock file.
153  1 */
154  1 (void) BsdDestroyPidFile(PIDPATH);
155  1 exit(signal);
156  1
157  } /* End of kill_handler() */
158  EDMDispatch.c 3
```

```
159  /*****
160  *
161  * Function Name:
162  * display_usage
163  *
164  * Simply displays the usage
165  *
166  * Call Arguments:
167  * program name
168  *
169  * Error Outputs and Side Effects:
170  * Prints usage.
171  *
172  * Special Considerations:
173  * None.
174  *****/
175  static void
176  display_usage (IN char *progname)
177  {
178  1 /* Print out usage stmt. */
179  1
180  1 fprintf(stderr, "Usage: %s [-d]\n", progname);
181  1 fprintf (
182  1 stderr, "-d keep the daemon from forking so debugging is easier\n");
183  1
184  1 } /* end display_usage () */
185  display_usage.c 4
```

```
187 /*****
188 **
189 ** Routine: daemon_catch_interrupts
190 **
191 ** Inputs:      None
192 **
193 ** Outputs:     None
194 **
195 ** Return Codes:
196 **      None
197 **
198 ** Purpose:     Sets up signals for service. On NT we will have to
199 **              consider what OS constructs to replace signals with.
200 **              In this case we are catching SIGTERM, SIGINT, and
201 **              SIGQUIT and ignoring anything else.
202 **
203 ** Intended caller: internal only.
204 **
205 *****/
206 */
207 void daemon_catch_interrupts()
208 {
209     struct sigaction  sactions;      /* Signal actions */
210
211     ZERO( sactions );
212
213     /*
214      * Set an empty list so we can set signals we want to handle
215      */
216     sigemptyset( &sactions.sa_mask );
217
218     /*
219      * Add signals that we want to handle
220      */
221     sigaddset( &sactions.sa_mask, SIGTERM );
222     sigaddset( &sactions.sa_mask, SIGINT );
223     sigaddset( &sactions.sa_mask, SIGQUIT );
224
225     /* Setup the signal handler. */
226     sactions.sa_handler = kill_handler;
227
228     /*
229      * Assign handler to each signal we are interested in.
230      */
231     (void) sigaction( SIGTERM, &sactions, NULL );
232     (void) sigaction( SIGINT, &sactions, NULL );
233     (void) sigaction( SIGQUIT, &sactions, NULL );
234
235     /*
236      * Setup mask so we can specify what signals we will ignore.
237      */
238     sigfillset( &sactions.sa_mask );
239
240     /*
241      * We want to ignore everything except those we have set up
242      * above so remove those from the list.
243      */
244     sigdelset( &sactions.sa_mask, SIGTERM );
245     sigdelset( &sactions.sa_mask, SIGINT );
246     sigdelset( &sactions.sa_mask, SIGQUIT );
247
248     /*
```

```
249     }
250     /* Set the mask. Since no other threads have been started,
251      * all threads will get this mask.
252      */
253     (void) thr_sigsetmask( SIG_SETMASK, &sactions.sa_mask, NULL );
254 }
```

```

256 /*****
257 **
258 ** Routine: daemon_check_proper_ID
259 ** Inputs:      None
260 ** Outputs:     None
261 ** Return Codes:
262 **             exits with an error when the user is not root
263 **
264 ** Purpose:
265 **         Checks user's ID and determines if the user is allowed
266 **         to execute service.
267 **         If there are no constraints then this
268 **         function may be blank.
269 **
270 ** Intended caller: internal only.
271 **
272 ****
273 ****
274 */
275 void daemon_check_proper_ID()
276 {
277     /*
278     ** Check for root
279     */
280     if (geteuid() != E_ROOTUID)
281     {
282         (void) EDMDispatch_logent(
283             FILE_, LINE_, LOG_ERR, DAEMON_NOTSUPERUSER, 0,
284             "Must be run as superuser, uid was %d",
285             geteuid());
286     }
287     exit(1);
288 }
289

```

```

291 /*****
292 **
293 ** Routine: parse_commandline
294 ** Inputs:      argc, argv (command line arguments)
295 ** Outputs:     None
296 ** Return Codes:
297 **             exits with an error when the user types a bad argument
298 **
299 ** Purpose:
300 **         Parses command line arguments and sets flags. If there
301 **         are no flags to be set then this function may be empty.
302 **
303 ** Intended caller: internal only.
304 **
305 ****
306 ****
307 */
308 void parse_commandline(int argc, char *argv[])
309 {
310     int opt; /* Process options */
311     commandlineargs = argv;
312     while ((opt = getopt(argc,argv,"d")) != EOF )
313     {
314         switch(opt)
315         {
316             case 'd':
317                 G_debug = TRUE;
318                 break;
319             default:
320                 (void) display_usage( argv[0] );
321                 exit(1);
322         }
323     }
324 }
325

```

```

332 /*****
333 **
334 ** Routine: daemon_initialize_logging
335 **
336 ** Inputs:      None
337 **
338 ** Outputs:     None
339 **
340 ** Return Codes:
341 **             None
342 **
343 ** Purpose:     Do whatever it takes to initialize logging. In the near
344 **             future this may involve doing something with catalogs or
345 **             calling higher level logging functions which encapsulate
346 **             these things.
347 **
348 ** Intended caller: internal only.
349 **
350 *****/
351 */
352
353 void
354 daemon_initialize_logging()
355 {
356     /* Pass in argv[0], the program name */
357     (void) esl_log_init(commandlineargs[0]);
358 }

```

```

360 /*****
361 **
362 ** Routine: daemon_become_daemon
363 **
364 ** Inputs:      None
365 **
366 ** Outputs:     None
367 **
368 ** Return Codes:
369 **             exits with an error code if initialization fails
370 **
371 ** Purpose:     This function is for doing the forking etc. under UNIX.
372 **             It is unknown what will be necessary under NT.
373 **
374 ** Intended caller: internal only.
375 **
376 *****/
377 */
378
379 void
380 daemon_become_daemon()
381 {
382     char *ptr;
383     int ret = 0;
384
385     /*
386      * Strip the path from the program name so we can use it
387      * elsewhere.
388      */
389     ptr = strrchr(commandlineargs[0], '/');
390     if (ptr == NULL)
391         ptr = commandlineargs[0];
392     else
393         ptr++;
394
395     /* Change directory to a process specific core directory */
396     ret = esl_coredir_setup(ptr);
397     if (ret != 0)
398     {
399         (void) EDMDispatch_logent( _FILE_, _LINE_, LOG_ERR,
400             MESSAGE_ERR_IN_ESL_COREDIR, 0,
401             "esl_coredir_setup failed: %d",
402             errno);
403         exit(1);
404     }
405
406     /*
407      * This is now esl functionality.
408      * to make this a "real" daemon by detaching from the
409      * changing the process group, closing stdout, stderr, stdin,
410      * ...
411      */
412     if (g_debug == FALSE)
413     {
414         ret = esl_daemon_startup();
415         if (ret != 0)
416         {
417             fprintf(
418                 stderr, "%s: Failed to initialize as daemon.\n",
419                 commandlineargs[0]);
420         }
421     }

```



```

419 3      }
420 2      }
421 1      }
422      }
exit(1);

```

```

424      /*****
425      **
426      ** Routine: rpc_init
427      **
428      ** Inputs:      None
429      **
430      ** Outputs:     None
431      **
432      ** Return Codes:
433      **      exits with an error code if initialization fails
434      **
435      ** Purpose:
436      **      This function is for doing RPC initialization.
437      **      For the most part it involves calling the csc routines.
438      **      This is pretty standard between UNIX and NT.
439      **
440      ** Intended caller: internal only.
441      *****/

```

```

442      */
443      void rpc_init()
444      {
445      error_status_t      status;
446      char      *ebuff;

```

```

449      /*
450      ** This is here because of HP which may or may not define timeval.
451      ** May be removed when esi_timeval is ported to clients
452      */
453      #ifdef STRUCT_TIMEVAL
454      struct timeval      sleep_interval = {5,0};

```

```

455      #else
456      struct timespec sleep_interval = {5,0};
457      /* 5 second sleep interval */

```

```

458      #endif
459      /* Setup the interface specification for RPC */
460      SERVER_IFSPEC(if_spec);

```

```

462      /*
463      * Login as SERVER_PRINCIPAL. The context of the process
464      * will be set to this principal.

```

```

465      * This process will keep trying to login to DCE if the
466      * registry
467      * server is unavailable.
468      * Note that under SUN RPC this is a no-op.

```

```

469      */
470      while (TRUE)
471      {
472      (void) csc_server_login(SERVER_PRINCIPAL,

```

```

473      SERVER_KEYTAB, &status);
474      /* If we succeeded, then exit this loop. */
475      if ( status == error_status_ok )

```

```

476      {
477      break;

```

```

478      }
479      else /* Print error message if appropriate. */
480      {
481      ebuff = (char *) csc_get_error( status );

```

```
483 3      (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
484 3      MESSAGE_NO_LOGIN, 0,
485 3      "CSC_SERVER_LOGIN failed: <%d>
486 3      %s",
487 2      status, (
488 2      ebuff ? ebuff : "Unknown error" ));
489 2      }
490 2      /* If the failure was due to unavailable client,
491 2      * pause and then try again.
492 2      */
493 2      if (status == sec_rgy_server_unavailable)
494 3      {
495 3          /*
496 3          * uses sleep when SUNRPC, otherwise uses
497 3          * pthread call to delay for the specified
498 3          * time
499 3          */
500 3          CSC_SLEEP(sleep_interval);
501 2          continue;
502 2      }
503 2      /* If we got here, we had a unexpected failure. */
504 2      (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
505 2      MESSAGE_NO_LOGIN, 0,
506 2      "The service cannot log in as
507 2      required");
508 2      }
509 1      exit(1);
510 1      }
511 1      /*
512 1      ** We need to initialize the authorization module before we
513 1      ** a listen.
514 1      */
515 1      (void) csc_authorization_init(&status);
516 1      if ( status != error_status_ok )
517 1      {
518 2          ebuff = (char *) csc_get_error( status );
519 2      }
520 2      (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
521 2      MESSAGE_NOAUTHORIZATION, 0,
522 2      "CSC_AUTHORIZATION_INIT failed: <%d> %s",
523 2      status, (
524 2      ebuff ? ebuff : "Unknown error" ));
525 2      exit(1);
526 1      }
527 1      (void) csc_register_server_interface( &if_spec,
528 1      SERVER_ANNOTATION,
529 1      &status);
530 1      if ( status != error_status_ok )
531 1      {
532 2          ebuff = (char *) csc_get_error( status );
533 2      }
534 2      (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
535 2      MESSAGE_CANNOTREGISTER, 0,
536 2      "CSC_REGISTER_SERVER_INTERFACE failed:
537 2      <%d> %s",
538 2      status, (
539 2      ebuff ? ebuff : "Unknown error" ));
540 2      exit(1);
```

```
541 2      }
542 1      exit(1);
543 1      ebuff ? ebuff : "Unknown error" ));
```

```

545 /*****
546 **
547 ** Routine: rpc_run
548 ** Inputs:      None
549 ** Outputs:     None
550 ** Return Codes:
551 **             None
552 ** Purpose:     This function is for running the RPC listen.
553 **             This is pretty standard between UNIX and NT.
554 ** Intended caller: internal only.
555 *****
556
557 */
558 void rpc_run()
559 {
560     error_status_t status;
561     /* error status (nbase.h) */
562     char *ebuff;
563     /* listen for RPC calls forever. */
564     (void) csc_server_listen(
565         rpc_c_listen_max_calls_default, &status );
566     ebuff = (char *) csc_get_error( status );
567     /* We don't expect to get here. */
568     (void) EDMDispatch_logent(
569         __FILE__, __LINE__, LOG_ERR, MESSAGE_SERVERLISTEN, 0,
570         "CSC_SERVER_LISTEN failed: <td> %s",
571         status, (
572             ebuff ? ebuff : "Unknown error" ) );
573 }
574
575
576
577
578

```

```

580 /*****
581 **
582 ** Routine: daemon_specific_initialization
583 ** Inputs:      None
584 ** Outputs:     None
585 ** Return Codes:
586 **             None
587 ** Purpose:     Do whatever makes this daemon special.
588 **             In some cases you
589 **             may want to start a thread or open a socket.
590 **             Do that here.
591 ** Intended caller: internal only.
592 *****
593
594 */
595 void
596 daemon_specific_initialization()
597 {
598     error_status_t status;
599     /* error status (nbase.h) */
600     int
601     pthread_t
602     pthread_t
603     pthread_t
604     pthread_t
605     pthread_t
606     pthread_t
607     pthread_t
608     pthread_t
609     pthread_t
610     pthread_t
611     pthread_t
612     pthread_t
613     pthread_t
614     pthread_t
615     pthread_t
616     pthread_t
617     pthread_t
618     pthread_t
619     pthread_t
620     pthread_t
621     pthread_t
622     pthread_t
623     pthread_t
624     pthread_t
625     pthread_t
626     pthread_t
627     pthread_t
628     pthread_t
629     pthread_t
630     pthread_t
631     pthread_t
632     pthread_t
633     pthread_t
634     pthread_t
635     pthread_t
636     pthread_t
637     pthread_t
638     pthread_t
639     pthread_t
640     pthread_t
641     pthread_t
642     pthread_t
643     pthread_t
644     pthread_t
645     pthread_t
646     pthread_t
647     pthread_t
648     pthread_t
649     pthread_t
650     pthread_t
651     pthread_t
652     pthread_t
653     pthread_t
654     pthread_t
655     pthread_t
656     pthread_t
657     pthread_t
658     pthread_t
659     pthread_t
660     pthread_t
661     pthread_t
662     pthread_t
663     pthread_t
664     pthread_t
665     pthread_t
666     pthread_t
667     pthread_t
668     pthread_t
669     pthread_t
670     pthread_t
671     pthread_t
672     pthread_t
673     pthread_t
674     pthread_t
675     pthread_t
676     pthread_t
677     pthread_t
678     pthread_t
679     pthread_t
680     pthread_t
681     pthread_t
682     pthread_t
683     pthread_t
684     pthread_t
685     pthread_t
686     pthread_t
687     pthread_t
688     pthread_t
689     pthread_t
690     pthread_t
691     pthread_t
692     pthread_t
693     pthread_t
694     pthread_t
695     pthread_t
696     pthread_t
697     pthread_t
698     pthread_t
699     pthread_t
700     pthread_t
701     pthread_t
702     pthread_t
703     pthread_t
704     pthread_t
705     pthread_t
706     pthread_t
707     pthread_t
708     pthread_t
709     pthread_t
710     pthread_t
711     pthread_t
712     pthread_t
713     pthread_t
714     pthread_t
715     pthread_t
716     pthread_t
717     pthread_t
718     pthread_t
719     pthread_t
720     pthread_t
721     pthread_t
722     pthread_t
723     pthread_t
724     pthread_t
725     pthread_t
726     pthread_t
727     pthread_t
728     pthread_t
729     pthread_t
730     pthread_t
731     pthread_t
732     pthread_t
733     pthread_t
734     pthread_t
735     pthread_t
736     pthread_t
737     pthread_t
738     pthread_t
739     pthread_t
740     pthread_t
741     pthread_t
742     pthread_t
743     pthread_t
744     pthread_t
745     pthread_t
746     pthread_t
747     pthread_t
748     pthread_t
749     pthread_t
750     pthread_t
751     pthread_t
752     pthread_t
753     pthread_t
754     pthread_t
755     pthread_t
756     pthread_t
757     pthread_t
758     pthread_t
759     pthread_t
760     pthread_t
761     pthread_t
762     pthread_t
763     pthread_t
764     pthread_t
765     pthread_t
766     pthread_t
767     pthread_t
768     pthread_t
769     pthread_t
770     pthread_t
771     pthread_t
772     pthread_t
773     pthread_t
774     pthread_t
775     pthread_t
776     pthread_t
777     pthread_t
778     pthread_t
779     pthread_t
780     pthread_t
781     pthread_t
782     pthread_t
783     pthread_t
784     pthread_t
785     pthread_t
786     pthread_t
787     pthread_t
788     pthread_t
789     pthread_t
790     pthread_t
791     pthread_t
792     pthread_t
793     pthread_t
794     pthread_t
795     pthread_t
796     pthread_t
797     pthread_t
798     pthread_t
799     pthread_t
800     pthread_t
801     pthread_t
802     pthread_t
803     pthread_t
804     pthread_t
805     pthread_t
806     pthread_t
807     pthread_t
808     pthread_t
809     pthread_t
810     pthread_t
811     pthread_t
812     pthread_t
813     pthread_t
814     pthread_t
815     pthread_t
816     pthread_t
817     pthread_t
818     pthread_t
819     pthread_t
820     pthread_t
821     pthread_t
822     pthread_t
823     pthread_t
824     pthread_t
825     pthread_t
826     pthread_t
827     pthread_t
828     pthread_t
829     pthread_t
830     pthread_t
831     pthread_t
832     pthread_t
833     pthread_t
834     pthread_t
835     pthread_t
836     pthread_t
837     pthread_t
838     pthread_t
839     pthread_t
840     pthread_t
841     pthread_t
842     pthread_t
843     pthread_t
844     pthread_t
845     pthread_t
846     pthread_t
847     pthread_t
848     pthread_t
849     pthread_t
850     pthread_t
851     pthread_t
852     pthread_t
853     pthread_t
854     pthread_t
855     pthread_t
856     pthread_t
857     pthread_t
858     pthread_t
859     pthread_t
860     pthread_t
861     pthread_t
862     pthread_t
863     pthread_t
864     pthread_t
865     pthread_t
866     pthread_t
867     pthread_t
868     pthread_t
869     pthread_t
870     pthread_t
871     pthread_t
872     pthread_t
873     pthread_t
874     pthread_t
875     pthread_t
876     pthread_t
877     pthread_t
878     pthread_t
879     pthread_t
880     pthread_t
881     pthread_t
882     pthread_t
883     pthread_t
884     pthread_t
885     pthread_t
886     pthread_t
887     pthread_t
888     pthread_t
889     pthread_t
890     pthread_t
891     pthread_t
892     pthread_t
893     pthread_t
894     pthread_t
895     pthread_t
896     pthread_t
897     pthread_t
898     pthread_t
899     pthread_t
900     pthread_t
901     pthread_t
902     pthread_t
903     pthread_t
904     pthread_t
905     pthread_t
906     pthread_t
907     pthread_t
908     pthread_t
909     pthread_t
910     pthread_t
911     pthread_t
912     pthread_t
913     pthread_t
914     pthread_t
915     pthread_t
916     pthread_t
917     pthread_t
918     pthread_t
919     pthread_t
920     pthread_t
921     pthread_t
922     pthread_t
923     pthread_t
924     pthread_t
925     pthread_t
926     pthread_t
927     pthread_t
928     pthread_t
929     pthread_t
930     pthread_t
931     pthread_t
932     pthread_t
933     pthread_t
934     pthread_t
935     pthread_t
936     pthread_t
937     pthread_t
938     pthread_t
939     pthread_t
940     pthread_t
941     pthread_t
942     pthread_t
943     pthread_t
944     pthread_t
945     pthread_t
946     pthread_t
947     pthread_t
948     pthread_t
949     pthread_t
950     pthread_t
951     pthread_t
952     pthread_t
953     pthread_t
954     pthread_t
955     pthread_t
956     pthread_t
957     pthread_t
958     pthread_t
959     pthread_t
960     pthread_t
961     pthread_t
962     pthread_t
963     pthread_t
964     pthread_t
965     pthread_t
966     pthread_t
967     pthread_t
968     pthread_t
969     pthread_t
970     pthread_t
971     pthread_t
972     pthread_t
973     pthread_t
974     pthread_t
975     pthread_t
976     pthread_t
977     pthread_t
978     pthread_t
979     pthread_t
980     pthread_t
981     pthread_t
982     pthread_t
983     pthread_t
984     pthread_t
985     pthread_t
986     pthread_t
987     pthread_t
988     pthread_t
989     pthread_t
990     pthread_t
991     pthread_t
992     pthread_t
993     pthread_t
994     pthread_t
995     pthread_t
996     pthread_t
997     pthread_t
998     pthread_t
999     pthread_t
1000    pthread_t

```

```
638 1      getrlimit(RLIMIT_NOFILE, &rlp);
640 1      (void) EDMDispatch_logent(
641 1          __FILE__, __LINE__, LOG_INFO, MESSAGE_STARTUP, 0,
        "Service allows %d open files",
        rlp.rlim_max );
        /* Initialize service launcher */
        ret = EDMDsvcsinit();
        if (ret != 0)
        {
            (void) EDMDispatch_logent(
                __FILE__, __LINE__, LOG_INFO, 0, 0,
                "Service launcher failed returning - %d",
                ret);
        }
        exit(1);
    }

    /*
     * Start the other threads in the daemon. The main thread
     * becomes the RPC thread. BAMdataManage is the entry point
     * for the data collection thread. BAMdataCleanup is the
     * entry point for the data expiration thread.
     */
    /* pthread_create(kmantid, NULL, DispDaemon_csr, NULL); */
    pthread_create(kmantid, NULL, DispDaemon_ccw, NULL);
    pthread_create(kcleantid, NULL, DispatchBackground, NULL);
    rpc_init();
    rpc_run();
}
```

```
666 1      /*****
667 1      **
668 1      ** Routine: daemon_cleanup
669 1      **
670 1      ** Inputs:      None
671 1      **
672 1      ** Outputs:     None
673 1      **
674 1      ** Return Codes:
675 1      **              None
676 1      **
677 1      ** Purpose:     Call function which will clean up daemon properly.
678 1      **
679 1      ** Intended caller: internal only.
680 1      **
681 1      *****/
682 1      */
        void
        daemon_cleanup()
        {
            kill_handler( 0 );
        }
    }
}
```


FreeSessionInfo	5	(EDMDispatchService.c)
cd_getServiceStatus_1_svc...	3	(EDMDispatchService.c)
cd_getSessionInfo_1_svc	4	(EDMDispatchService.c)
dd_initialize_1_svc.....	2	(EDMDispatchService.c)


```
1  /*
2  ** Copyright 1996, 1997 EMC Corporation
3  */
4
5  /*
6  * EDMDispatchService.c
7  *
8  * Mission Statement: RPC entry points.
9  * Primary Data Acted On:
10 *
11 * Compile-Time Options:
12 *
13 * Basic idea here:
14 */
15
16
17 #if defined(lint)
18 static char RCS_id [] = "@(#)$RCSfile: EDMDispatchService.c,v $"
19 "$Revision: 1.0 $"
20 "$Date: 1997/02/06 20:49:15 $" ;
21
22 #endif
23
24 #include <esl/c_portable.h>
25 #include <esl/inout.h>
26
27 #include <logging/logging.h>
28 #include <csc/csccomm.h>
29
30 #include <restore/csc_EDMDispatch.h>
31 #include <restore/dispatch_daemon.h>
32
33 #include <EDMDispatchlog.h>
34 #include <EDMDispatchSession.h>
35
36 /*
37 * These are all the rpc entry points for the dispatch daemon.
38 * The dispatch daemon is multi-threaded and it is the main thread
39 * which handles all incoming RPC. ONC RPC is single threaded
40 * so each call blocks other RPC calls. This provides us some
41 * safety in the way we handle our data and limits our exposure
42 * to unexpected multithreading problems.
43 */
44 static void FreeSessionInfo(SessionInfo *);
45
46 /*****
47 **
48 ** Routine: dd_initialize_1
49 **
50 ** Inputs:  DD_initialize_args * - args for the restore initialize
51 **          call
52 **
53 ** Outputs: None
54 **
55 ** Return Codes:
56 **      DD_initialize_result * - result of init function call
57 **
58 ** Purpose: Function to create a restore session.
59 **
60 ** Intended caller: Internal Only.
61 **
62 *****/
63
64 DD_initialize_result *
```

```
64 dd_initialize_1_svc(
65 1 {
66 1     static DD_initialize_result argzz;
67 1     InitializeSession(arg, req, kargzz);
68 1     return kargzz;
69 1 }
70
71 }
```



```
73 /*****
74 **
75 ** Routine: dd_getservicestatus_1
76 **
77 ** Inputs: DD_getservicestatus_args * - args for the
78 **          getservicestatus call
79 **
80 ** Outputs: None
81 **
82 ** Return Codes:
83 **              DD_getservicestatus_result * - result of status function
84 **              call
85 **
86 ** Purpose: Function to poll for status on a session.
87 **
88 ** Intended caller: Internal Only.
89 *****/
```

```
88 */
89
90 DD_getservicestatus_result *
91 dd_getservicestatus_1_svc(
92     IN DD_getservicestatus_args *arg, IN struct svc_req *req )
93 {
94     static DD_getservicestatus_result argzz;
95
96     GetDispatchStatus(arg, &argzz);
97
98     return &argzz;
99 }
```

```
100 /*****
101 **
102 ** Routine: dd_getsessioninfo_1
103 **
104 ** Inputs: DD_getservicestatus_args * - args for the getsessioninfo
105 **          call
106 **
107 ** Outputs: None
108 **
109 ** Return Codes:
110 **              SessionBlock * - result of session info call
111 **
112 ** Purpose: Function to get information on all sessions.
113 **
114 ** Intended caller: Internal Only.
115 *****/
```

```
117 SessionBlock *
118 dd_getsessioninfo_1_svc(
119     IN DD_getservicestatus_args *arg, IN struct svc_req *req )
120 {
121     static SessionBlock argzz;
122     static boolean_t first = TRUE;
123
124     if (first)
125     {
126         memset(&argzz, 0, sizeof(argzz));
127         first = FALSE;
128     }
129     else
130     {
131         FreeSessionInfo(argzz.sess);
132         argzz.sess = NULL;
133     }
134
135     GetDispatchInfo(arg, &argzz);
136
137     return &argzz;
138 }
```

```

139  /*****
140  **
141  ** Routine: FreeSessionInfo
142  **
143  ** Inputs: SessionInfo * - arg to free
144  **
145  ** Outputs: None
146  **
147  ** Return Codes:
148  **      None
149  **
150  ** Purpose: Function to free all SessionInfo structures in a list.
151  **
152  ** Intended caller: Internal Only.
153  *****/
154
155  static void FreeSessionInfo(SessionInfo *sess)
156  {
157      if (sess == NULL)
158          return;
159
160      if (sess -> next != NULL)
161          FreeSessionInfo(sess -> next);
162
163      free(sess);
164  }

```

